ECE 438: Computer Design
Electrical and Computer Engineering
University of New Mexico

Instructor: Andrew Targhetta

Lab 1

The Register File and ALU

Due: March 5th, 2020

Introduction

This lab is the first of many labs that will guide you through the design of a MIPS processor using a well-know Hardware Description Language (HDL). The objective of the first lab is to provide a refresher on HDL development by describing the *register file* and *Arithmetic Logic Unit (ALU)* in the VHSIC-Hardware Description Language (VHDL). In subsequent labs, we will use VHDL to integrate the register file and ALU into a complete datapath. Although the laboratory exercises are VHDL centric, you are certainly welcome to use Verilog, an alternative HDL that continues to gain widespread industry and academic support.

Background

The MIPS Instruction Set Architecture (ISA) is part of the Reduce Instruction Set Computer (RISC) design philosophy that simplifies instruction decoding with a small number of fixed-width instruction formats coupled with the *load-store* architecture concept. In a load-store architecture, data must first be *loaded* into the General Purpose Registers (GPRs) from memory before it can be manipulated using the ALU. After data manipulation, it can then be *stored* back out to memory.

The MIPS 32-bit ISA only allows for at most two 32-bit registers to be read and one 32-bit register to be written per instruction. Thus, the register file only needs two read ports and one write port. Similarly, the ISA only allows computation on at most two operands so the ALU has two 32-bit inputs and one 32-bit result. For the sake of simplicity, we do not support multiplication or division, which could produce more than 32 bits of result data. Figure 1 depicts the inputs and outputs of two components you will describe in this lab. The blue lines signify control signals while the black lines signify data. The thicker lines represent multiple bit vectors while the thinner lines represent single bit signals.

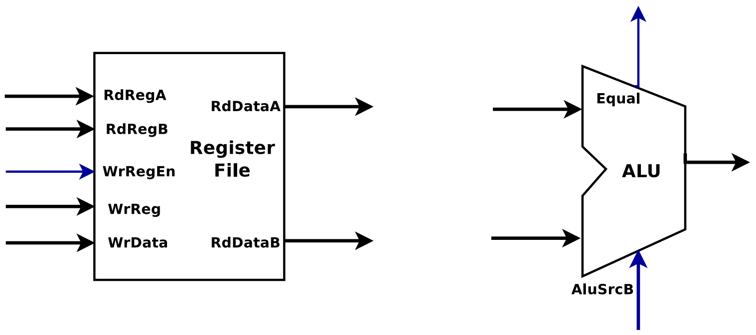

Figure 1: The register file and ALU

A processor is made up of a mix of *combinational* and *sequential* logic. In the case of combinational logic, the output is dependent solely on the inputs, whereas in the case of sequential logic, the output depends on the inputs and the state. The ALU is an example of purely combinational logic, i.e. it has no state. The register file on the other hand has state. In fact, storing processor state is the register file's primary purpose! In VHDL, we can describe combinational logic with simple assignment statements or with a *process* block such that all of the inputs to our combinational block are listed in the *sensitivity list*. For sequential logic, we will use a *process* block with just the clock in the sensitivity list.

Procedure

1. Describe the **Register File** in VHDL as a small memory with two read ports and a write port. In our case, the read ports are asynchronous, meaning the data on the output ports, **RdDataA** and **RdDataB**, do not wait until a clock edge to change, but rather change some propagation delay after the associated register address ports, **RdRegA** and **RdRegB**, change. Note that we could use VHDL to model this propagation delay, but our goal here is to create *synthesizable* code. This means that we can take the code described in lab and *synthesize* to an FPGA target. Although our simulations will show these asynchronous signals changing instantaneously, know that when your code is synthesized and mapped to a target technology, there will be a propagation delay.

   Writes into the register file happen on the *rising edge* of the clock. Note that we will change this to the *falling edge* in future labs when we pipeline our processor. Finally, our register file for MIPS must keep the zero register zeroed and not allow writes to it. The code below describes most of the register file design but omits a few key parts.

```
---------------------------------------------------
-- This source file describes a 32x32 register file such
-- that the 0th register is always 0 and is not writeable
---------------------------------------------------

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

---------------------------------------------------

entity RegisterFile is

port( RdRegA: in std_logic_vector(4 downto 0);
      --insert code here for read port B
      WrReg: in std_logic_vector(4 downto 0);
      Clk: in std_logic;
      --something else is missing here...
      WrData: in std_logic_vector(31 downto 0);
      RdDataA: out std_logic_vector(31 downto 0);
      --insert code here for read port B
);
end RegisterFile;

---------------------------------------------------

architecture RegisterFile of RegisterFile is

---------------------------------------------------
    --create an array of 31 32-bit registers
    type register_array is array (1 to 31) of
        std_logic_vector (31 downto 0);
    signal Registers: register_array;
---------------------------------------------------

begin

---------------------------------------------------
--describe the write functionality
    process(Clk)--only do something if clock changes
    begin
        --on the rising edge of clock
        if(Clk'event and Clk='1') then
            --only write if enabled and not
            --attempting to write to 0 reg
            if(RegWrEn='1' and conv_integer(WrReg)/=0) then
                Registers(conv_integer(WrReg)) <= --finish this line...
            end if;
        end if;
    end process;
---------------------------------------------------
--describe the read functionality
    process(RdRegA, Registers)
    begin
        if(conv_integer(RdRegA)=0) then
            RdDataA <= (others => '0');
        else
            RdDataA <= Registers(conv_integer(RdRegA));
        end if;
    end process;

    --insert code here for B port!

end RegisterFile;
---------------------------------------------------
```

2. Describe the ALU in VHDL. **The ALU** is a combinational logic unit that produces an arithmetic or logic result from two input operands based on the ALU control bus. The two operand buses, *AluInA* and *AluInB*, are each 32-bits wide, while the control bus is 4-bits wide. The outputs from the ALU module include a 32-bit result, *AluResult*, and an *Equals* signal, which is asserted when the input operands are equal. Note that we are not supporting exceptions in this lab, so there is no need to detect arithmetic overflow. The ALU can be thought of as a combination of arithmetic and logic blocks, each sharing the same inputs, such that the outputs of each of the blocks are multiplexed to create the final result. In this illustration, the ALU control signals act as the selection lines of the multiplexer. Table 1 list the ALU operations you must support along with their associated control codes. The follow entity should be used for your ALU:

```
---------------------------------------------------
-- Put some comments here describing the ALU
---------------------------------------------------

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all; --needed to describe
use ieee.std_logic_arith.all;    --arithmetic on std_logic_vector
use ieee.numeric_std.all;        --types. Treated as unsigned!
entity ALU is

port( AluCtrl: in std_logic_vector(3 downto 0);
      AluInA, AluInB: in std_logic_vector(31 downto 0);
      AluResult: out std_logic_vector(31 downto 0);
      Equals: out std_logic
);
end ALU;
```

Opperation	ALU Control Line
AND	0000
OR	0001
SLL	0011
SRL	0100
ADDU	1000
SUBU	1001
XOR	1010
SLTU	1011
NOR	1100
SRA	1101
LUI	1110

Table 1: Arithmetic Logic functions and associated control codes

Helpful Hints:

- In VHDL, the CASE statement makes the ALU descriptions quite simple and can be expanded easily later on.
- Be sure to include a "WHEN OTHERS" statement in your description to avoid create latches.
- Many simulators do not support newer versions of VHDL that allow direct shift operations on `std_logic_vector` types. Consequently, you may have to perform some type conversions to use the `sll`, `srl`, and `sra` operators. For example:

```
AluResult <= to_stdlogicvector(to_bitvector(AluInB) sll conv_integer(
    AluInA));
```

In this case, `sll` requires that the shift amount be an `integer` and the shift value be a `bit_vector`. Likewise, the result of `sll` is of type `bit_vector`. To use the above conversion functions, you must include the `ieee.numeric_std.all` library.

1 Deliverables

1. Email all VHDL source files with comments to `adtargh@unm.edu` .
   Note: Code submitted without adequate comments will not be graded!

2. Answer the following questions:

(a) Two types of port maps can be used when instantiating a component in VHDL, namely positional and nominal. Describe the difference between these two port map styles. Which do you think would be preferred for processor design and why?

(b) How many flip-flops does our register file need when synthesized? How many flip-flops does our VHDL description of the register file imply?

(c) Which VHDL simulator(s) have you used in the past.

(d) What sources do you use as VHDL reference guides? Why?